METHOD, APPARATUS, AND COMPUTER PROGRAM PRODUCT FOR IMPLEMENTING ENHANCED DEBUGGER GRAPHICAL USER INTERFACE FUNCTIONS

Field of the Invention

The present invention relates generally to the data processing field, and more particularly, relates to a method, apparatus, and computer program product for implementing enhanced graphical user interface functions in a graphical debugger.

Description of the Related Art

An important aspect of the design and development of a computer program is a process known as debugging. A computer programmer to locate and identify errors in a program under development performs debugging. Typically, a programmer uses another computer program commonly known as a debugger to debug a program under development.

A problem with known debuggers is that some useful and often needed information for a user is not available from a graphical user interface (GUI) of the available debuggers. Most Unix style debuggers display a flat list of source files that are in the program being debugged. Although this is very useful, sometimes the programmer needs additional data that cannot be displayed with known debuggers.

For example, some programs can dynamically load object files and object archives at run time. For programs that dynamically bind to program objects, the information about these objects is stored in what is commonly called a loadmap. Today the only other way to discern this information would be to examine make files, or to dump the debug data of the individual object files or archives with an external dumping tool.

A need exists for an improved debugger having a mechanism when debugging such programs enabling display for the programmer of what objects are in each part of the loadmap and the source files that correlate to these objects.

Another problem with known debuggers is that when records or variable structures are displayed, either all or none of the fields of the records are displayed by the conventional graphical user interface (GUI) of the available debuggers. Often the programmer or user is interested only one or a subset of the fields that are displayed. For example, consider an abstract sting type that may contain a field for the length, the character encoding, and a pointer to the string buffer. During debug sessions, the user often is only interested in the contents of the string buffer.

A need exists for a an improved debugger having a mechanism when debugging programs to provide custom record displays, where only user selected fields of records or variables are displayed.

Summary of the Invention

One aspect of the present invention is to provide a method, apparatus, and computer program product for implementing enhanced graphical user interface functions in a graphical debugger. Other important aspects of the present invention are to provide such method, apparatus and computer program product substantially without negative effect and that overcome many of the disadvantages of prior art arrangements.

In brief, a method, apparatus and computer program product are provided for implementing enhanced graphical user interface functions in a graphical debugger. A user interface operatively controls a graphical user interface. A loadmap display manager coupled to the user interface implements a loadmap function. The user interface responsive to the loadmap display manager displays a program loadmap. A custom record display manager coupled to the user interface receives user inputs and implements a custom record display function. The user interface responsive to the custom record display manager displays user selected customized records.

In accordance with features of the invention, to generate the program loadmap, the loadmap display manager reads debug data for a program under debug. The loadmap display manager examines the program debug data, generates a list of each source and disassembly file that the program contains, and generates lists of each object or archive program bound to the program under debug at run time. The generated program loadmap is sent to the GUI for display. The loadmap display manager identifies program load and unload events, and dynamically updates the program loadmap display as the loadmap information changes. Using the program loadmap display of the invention enables setting a breakpoint within a user selected address range or one instance of a source file, without setting the breakpoint in other instances of the source file.

In accordance with features of the invention, custom record display manager identifies a user selected variable, and user selected fields of the variable to be displayed. If the user selects all variables of this type to be customized, then a custom record with the user selected fields is created and added to a custom type list. Then all variables of this type are displayed only with the user selected fields. Otherwise, a custom record with the user selected fields is created and added to a variables list and the variable is displayed only with the user selected fields.

Brief Description of the Drawings

The present invention together with the above and other objects and advantages may best be understood from the following detailed description of the preferred embodiments of the invention illustrated in the drawings, wherein:

FIGS. 1A and 1B are block diagram representations illustrating a computer system and operating system for implementing an enhanced user interface in a graphical debugger in accordance with the preferred embodiment;

FIGS. 2A, 2B and 3 are flow charts illustrating exemplary steps for implementing a loadmap function of the enhanced user interface in accordance with the preferred embodiment;

FIG. 4 illustrates an exemplary graphical user interface including the loadmap function in accordance with the preferred embodiment;

FIGS. 5-10 are flow charts illustrating exemplary steps for implementing a custom structure display function of the enhanced user interface in accordance with the preferred embodiment;

FIG. 11 illustrates an exemplary graphical user interface including setup options for the custom structure display function in accordance with the preferred embodiment;

FIG. 12 illustrates an exemplary graphical user interface including an exemplary custom structure display function resulting from the illustrated user selection of FIG. 11 in accordance with the preferred embodiment; and

FIG. 13 is a block diagram illustrating a computer program product in accordance with the preferred embodiment.

Detailed Description of the Preferred Embodiments

Referring now to the drawings, in FIGS. 1A and 1B there is shown a computer system generally designated by the reference character 100 for implementing enhanced graphical user interface functions in a graphical debugger in accordance with the preferred embodiment. Computer system 100 includes a main processor 102 or central processor unit (CPU) 102 coupled by a system bus 106 to a memory management unit (MMU) 108 and system memory including a dynamic random access memory (DRAM) 110, a nonvolatile random access memory (NVRAM) 112, and a flash memory 114. A mass storage interface 116 coupled to the system bus 106 and MMU 108 connects a direct access storage device (DASD) 118 and a

CD-ROM drive 120 to the main processor 102. Computer system 100 includes a display interface 122 connected to a display 124, and a network interface 126 coupled to the system bus 106.

Computer system 100 is shown in simplified form sufficient for understanding the present invention. The illustrated computer system 100 is not intended to imply architectural or functional limitations. The present invention can be used with various hardware implementations and systems and various other internal hardware devices, for example, multiple main processors.

As shown in FIG. 1B, computer system 100 includes an operating system 130, a computer program 132, and a source level debugger 134. The source level debugger 134 includes a user interface 136 coupled to and operatively controlling an enhanced graphical user interface (GUI) 138 of the preferred embodiment, a loadmap display manager 140 of the preferred embodiment, a custom record display manager 142 of the preferred embodiment, and a debugger server 144. While separately shown, the loadmap display manager 140 and custom record display manager 142 could be implemented as integral functions within the debugger server 144. The enhanced graphical user interface (GUI) 138 operatively controlled by the user interface 136 displays loadmap and custom record features of the preferred embodiment.

In accordance with features of the preferred embodiment, loadmap display manager 140 implements a new loadmap function and a new loadmap tab, such as loadmap tab 402 illustrated and described with respect to FIG. 4, is added to the enhanced GUI 138 to display a program loadmap and dynamically update the program loadmap display as the program loadmap changes as programs are loaded or unloaded. Custom record display manager 142 allows the user to program the user interface 136 to remember which fields of a given variable type or variable to be sent to the enhanced GUI 138 for display. Then whenever a variable of the particular type or the record is encountered, the enhanced GUI 138 initially displays only the user-selected or user-programmed fields for the variable or the record.

Various commercially available computers can be used for computer system 100; for example, an eServer ® iSeries ® computer system manufactured and sold by International Business Machines Corporation and processor 102 can be implemented, for example, by one of a line of IBM ® PowerPC ® processors manufactured and sold by International Business Machines Corporation. Central processor unit 102 is suitably programmed to execute the flowchart of FIGS. 2A, 2B, and 3 to generate a loadmap function and generate the enhanced GUI loadmap display screen of FIG. 4 of the preferred embodiment and to execute the flowchart of FIGS. 5-10 to generate custom display records and generate the enhanced GUI custom record display screens of FIGS. 11 and 12 of the preferred embodiment.

In accordance with features of the preferred embodiment, the debugger enhanced GUI 138 shows the topography of the program and which source files have been bound into which objects. The loadmap function is extremely important when debugging code that the user did not write or that is not familiar to the user. Unix allows binding modules with the same name into several different archives; and to have the same name source file appear in different modules or archives. These sources files may or may not contain the same source code. Thus it is very important to be able to graphically sort out which source files appear in which archives and object files, as provided by the debugger GUI 138 of the preferred embodiment.

In accordance with features of the preferred embodiment, the loadmap display manager 140 operatively controls the debugger server 142 for examining the program debug data to get lists of all the object files and archives to which the program under debug is bound. The loadmap display manager 140 operatively controls the debugger server 142 to make a list of each source and disassembly file that the program contains. This information is sent back to the user interface 136 for display on the enhanced GUI 138 of the preferred embodiment. The loadmap display manager 140 operatively controls the debugger server 142 to listen for program load and unload events, and dynamically send current loadmap information to the user interface 136 for display on the enhanced GUI 138 as the loadmap information changes.

Referring now to FIGS. 2A, 2B and 3, there are shown exemplary steps for implementing a loadmap function of loadmap display manager 140 to generate the enhanced graphical user interface 138 in accordance with the preferred embodiment.

In FIG. 2A, a computer program 132 is placed under debugger 134 as indicated in a block 200. Debug data for the main program 132 under debug are read as indicated in a block 202. For each object, .O, in the main program as indicated in a block 204 processing is performed to identify address ranges for each source file and disassembly file of the main program. Checking for main start initialized is performed as indicated in a decision block 206. If main start is not initialized, main start is set to .O start as indicated in a block 208. If main start is initialized, checking whether .O start is less than main start as indicated in a decision block 210. If .O start is less than main start, then main start is set to .O start as indicated in a block 208. After setting main start to .O start at block 208 or 210, and when .O start is not less than main start, then checking whether main end is initialized as indicated in a decision block 214. If main end is not initialized, main end is set to .O end as indicated in a block 216. When main end is initialized, then checking whether O end is greater than main end is performed as indicated in a decision block 218. If .O end is greater than main end, then main end is set to .O end as indicated in a block 220. When .O end is not greater than main end or after main end is set to .O end at block 216 or 220, then a source file name is cached as indicated in a block 222. An entry representing disassembly for the main program is cached as indicated in a block 224 and a next object in the main program is processed. Then sequential operations continue following entry point A in FIG. 2B to add information to the loadmap of all other objects or archive programs bound to the main program 132 at run time.

In FIG. 2B, as indicated in a block 226 for each archive program linked to by the main program, debug data are read as indicated in a block 228. For each object, .O, in the archive program as indicated in a block 230 processing is performed to identify address ranges for each archive program. Checking for archive start initialized is performed as indicated in a decision block 232. If archive start is not initialized, archive start is set to .O start as indicated in a block 234. If archive start is initialized, checking whether .O start is less than archive start as indicated in a decision block 236. If .O start is less than archive start, then archive start is set to .O start as indicated in a block 238. After setting archive start to .O start at block 234 or 238, and when .O start is not less than archive start, then checking whether archive end is initialized as indicated in a decision block 240. If archive end is not initialized, archive end is set to .O end as indicated in a block 242. When archive end is initialized, then checking whether O end is greater than archive end is performed as indicated in a decision block 244. If .O end is greater than archive end, then archive end is set to .O end as indicated in a block 246. Then after archive end is set to .O end at block 242 or 246, and when .O end is not greater than archive end, then a source file name is cached as indicated in a block 248. An entry representing disassembly for the archive program is cached as indicated in a block 250 and a next object in the archive program is processed. Cached information is sent to GUI 142 as indicated in a block 252.

Referring to FIG. 3, breakpoints are set in a program 132 under test at each load and unload entry point as indicated in a block 300. Upon execution of a program 132 under test, processing of that program is temporarily suspended when the load or unload breakpoint instruction is executed. Debugging the program is started as indicated in a block 302. A load or unload breakpoint is hit as indicated in a block 304. Checking whether the breakpoint is a load breakpoint is performed as indicated in a decision block 306. If a load breakpoint is not identified, then all information about this archive being unloaded is removed from cache as indicated in a block 308. Then the GUI 138 is notified of the archive removal as indicated in a block 310. If a load breakpoint is identified, then all information about this archive being loaded is gathered and cached as indicated in a block 312. Then the cached information is sent to GUI 138 as indicated in a block 314. The program terminates as indicated in a block 316.

Referring now to FIG. 4, there is shown an exemplary graphical user interface screen generally designated by reference character 400 of the enhanced GUI 138 including the loadmap function of loadmap display manager 140 in accordance with the preferred embodiment. The illustrated exemplary GUI screen 400 includes a loadmap tab 402 of the preferred embodiment that provides a tree format generally indicated by 404 of each object 406 that the loadmap contains. Under the individual object 406 is a list of corresponding source files 408, such as testpgm.C, and a disassembly file 408, such as testpgm.32, that make up that object. Enhanced GUI 138 also displays a source code 412 and a disassembly code 414 of each object 406 and displays an address range associated with the source files 408. With the click of a mouse the user can display source code 412 of any of the source files 408 associated with the loadmap objects. The displayed loadmap information of enhanced GUI 132 enables a user to set a breakpoint within the displayed instance of source code 412, without setting a breakpoint in other instances of the source file 408.

Referring now to FIGS. 5-10, there are shown flow charts illustrating exemplary steps for implementing a custom record display function of custom record display manager 142 to generate the enhanced graphical user interface 138 in accordance with the preferred embodiment.

In FIG. 5, sequential operations of a main GUI process start at a block 500. An event is identified as indicated in a block 502. Checking whether the event is to display panel is performed as indicated in a decision block 504. If so, then a display panel routine is performed as indicated in a block 506. The display panel routine is illustrated and described with respect to FIG. 7. Otherwise, checking whether the event is to expand all is performed as indicated in a decision block 508. If the event is to expand all, then an expand all routine, illustrated and described with respect to FIG. 9, is performed as indicated in a block 510. In event is not to expand all, then checking whether the event is to collapse all is performed as indicated in a decision block 512. If the event is to collapse all, then a collapse all routine, illustrated and described with respect to FIG. 8, is performed as indicated in a block 514. In event is not to collapse all, then checking whether the event is a user expand or collapse is performed as indicated in a decision block 516. If the event is a user expand or collapse, then a user expand/collapse routine, illustrated and described with respect to FIG. 10, is performed as indicated in a block 518. In event is not a user expand or collapse, then checking whether the event is to a create custom record is performed as indicated in a decision block 520. If the event is to create custom record, then a create custom routine, illustrated and described with respect to FIG. 6, is performed as indicated in a block 522. If the event is not to create custom structure, then other events are processed as normal.

In FIG. 6, there are shown sequential operations of a create custom process of the preferred embodiment starting at a block 600. A variable is selected by the user as indicated in a block 602. Then the user selects fields for the variable to be displayed as indicated in a block 604. Checking whether the selected variable is associated with a type of variable is performed as indicated in a decision block 606. If the selected variable is associated with a type of variable, then each variable of this type is marked as not previously visited as indicated in a block 610. Then a custom record with the user selected fields is created as indicated in a block 612. The custom record is added to a custom type list as indicated in a block 614. Otherwise, if the selected variable is not associated with a type of variable, then the variable is marked as not previously displayed as indicated in a block 614. Then a custom record with the selected fields is created as indicated in a block 616. The custom record is added to a variable list as indicated in a block 618 and the sequential process is completed as indicated in a block 620.

In FIG. 7, sequential operations of a display panel process start at a block 700 and display panel content are identified as indicated in a block 702. Then processing is performed for each entry in the display panel as indicated in a block 604 that starts with checking whether the entry has children as indicated in a decision block 706. If the entry does not have children, then the variable is displayed. If the entry has children, checking whether the variable was previously visited or displayed is performed as indicated in a decision block 710. If previously displayed, then the variable is displayed as previously displayed as indicated in a block 710. Otherwise if not previously displayed, then checking for a custom record for the variable is performed as indicated in a decision block 714. If the variable has a custom record, then the variable is displayed only with the fields in the variable custom record as indicated in a block 716. If no custom record for the variable is identified, then checking whether a custom type for the variable exists as indicated in a decision block 718. If a custom type for the variable exists, then the variable is displayed only with the user selected fields in the variable type custom record as indicated in a block 720. Otherwise, if no custom type for the variable exists, then the variable is displayed in default manor or collapsed as indicated in a block 722.

In FIG. 8, sequential operations of a collapse all process start at a block 800 and display panel content are identified as indicated in a block 802. Then processing is performed for each entry in the display panel as indicated in a block 804 including checking whether the entry has children is performed as indicated in a decision block 806. If the entry does not have children, then the variable is displayed as indicated in a block 808. If the entry has children, then the entry is displayed collapsed as indicated in a block 810.

In FIG. 9, sequential operations of an expand all process start at a block 900 and display panel content are identified as indicated in a block 902. Then processing is performed for each entry in the display panel as indicated in a block 904 including checking whether the entry has children is performed as indicated in a decision block 906. If the entry does not have children, then the variable is displayed as indicated in a block 908. If the entry has children, then the entry is displayed fully expanded as indicated in a block 910.

In FIG. 10, sequential operations of a expand/collapse process start at a block 1000 checking whether the record is collapsed is performed as indicated in a decision block 1002. If the record is collapsed, then checking whether a custom record exists for the variable as indicated in a decision block 1004. If a custom record exists for the variable, then the record is expanded according to the custom record as indicated in a block 1006 and the sequential operations are completed as indicated in a block 1007. If a custom record does not exist for the variable, then checking whether a custom type record exists for the variable as indicated in a decision block 1010. If a custom type record exists for the variable, then the record is expanded according to the custom type record as indicated in a block 1012 to complete the sequential operations at block 1007. Otherwise, if no custom type record exists for the variable, then the record is fully expanded as indicated in a block 1112 to complete the sequential operations at block 1007. If determined at decision block 1002 that the record is not collapsed, then checking whether the record is fully expanded as indicated in a decision block 1014. If the record is fully expanded, then the record is collapsed as indicated in a block 1016 to complete the sequential operations at block 1007. Otherwise if the record is fully expanded, then the record is fully expanded as indicated in a block 1018 to complete the sequential operations at block 1007.

Referring now to FIG. 11, there is shown an exemplary graphical user interface screen generally designated by reference character 1100 of GUI 138 including setup options for the custom record display function of the custom record display manager 142 in accordance with the preferred embodiment. To initiate the custom structure display function the user first selects a variable of the type to be customized or programmed, for example, using a context menu selects a Type Setup option. For example, the user selects a variable 1102, shown as variable foo which is a type of testClass. Then as shown in the exemplary graphical user interface screen 1100 of GUI 138 each variable has a name 1104 and value 1006, and the selected variable foo, 1102 is displayed fully expanded with multiple fields 1008 and check boxes 1008 appear next to each of the fields of the variable structure. The user checks the fields via check boxes 1008 that are to be displayed for the selected variable 1102, and then, for example, opens the context menu again and the option on the context menu says Apply Type Setup. After this Apply Type Setup option is selected all variables of the type, such as testClass, will be displayed by the rules just established.

Referring now to FIG. 12, there is shown an exemplary graphical user interface screen generally designated by reference character 1200 of GUI 138 including an exemplary custom structure display function resulting from the illustrated user selection of FIG. 11 in accordance with the preferred embodiment. As shown, the selected variable foo, 1102 is displayed with the value 1106 of only field 12, x, 1008 displayed.

In addition to selecting which fields 1008 to display the programmer can also override the field name, or specify that a single value should be displayed after the structure. Also, any options, such as ASCII or HEX, that are active when the Apply Type Setup option is selected are remembered and used for other variables of this type. When base class members are included in a variable expansion, and a variable of the base classes type has been customized or programmed, only the fields specified at that time will be included.

The user can override this for the particular type by programming that type to include or exclude more of the base class fields. This programmed information can also be saved in an environment file to be used each time the debugger 134 is run.

Referring now to FIG. 13, an article of manufacture or a computer program product 1300 of the invention is illustrated. The computer program product 1300 includes a recording medium 1302, such as, a floppy disk, a high capacity read only memory in the form of an optically read compact disk or CD-ROM, a tape, a transmission type media such as a digital or analog communications link, or a similar computer program product. Recording medium 1302 stores program means 1304, 1306, 1308, 1310 on the medium 1302 for carrying out the methods for implementing the enhanced graphical user interface features of the preferred embodiment in the system 100 of FIG. 1.

A sequence of program instructions or a logical assembly of one or more interrelated modules defined by the recorded program means 1304, 1306, 1308, 1310, direct the computer system 100 for implementing the enhanced graphical user interface features of the preferred embodiment.

While the present invention has been described with reference to the details of the embodiments of the invention shown in the drawing, these details are not intended to limit the scope of the invention as claimed in the appended claims.